AMENDMENTS TO SPECIFICATION

Please amend the Title as follows:

SYSTEM AND METHOD TO REDUCE THE SIZE OF ~~EXECUTABLE~~ SOURCE CODE IN A PROCESSING SYSTEM

Please amend the following paragraphs of the Specification as follows:

[0001] The present invention relates generally to compiler systems and, more particularly, to a system and method to reduce the size of ~~executable~~source code in a processing system.

[0002] Increasingly, the size of compiled code has an impact on the performance and economics of computer systems. From embedded systems, such as cellular phones, to applets shipped over the World Wide Web, the impact of compile-time decisions that expand the size of the ~~executable~~source code has a direct effect on cost and power consumption, as well as on the transmission and execution time of the code.

[0003] Several techniques have been proposed to reduce the size of the ~~executable~~source code. One known technique turns repeated code fragments into procedures and is usually applied to intermediary code or even source code. However, this technique appears to miss repeated code fragments introduced during code generation. Another known technique reuses the common tail of two merging code sequences and is usually performed after code generation. However, syntactic mismatches seem to affect the efficiency of this technique.

[0005] **Figure 1** is an exemplary ~~executable~~source code within a program.

[0006] **Figure 2A** is a block diagram of one embodiment of a graph structure representing the ~~executable~~source code shown in **Figure 1**.

[0007] **Figure 2B** is a block diagram of one embodiment of the graph structure showing a reduced ~~executable~~source code obtained through a method to reduce the size of the ~~executable~~source code in a processing system.

[0008] **Figure 3** is a flow diagram of one embodiment of the method to reduce the size of the ~~executable~~source code.

[0009] **Figure 4A** is a block diagram of one embodiment of an interference graph structure constructed in connection with the ~~executable~~source code.

[0011] **Figure 5** is a block diagram of a data dependence graph structure constructed in connection with the ~~executable~~source code.

[0012] **Figure 6** is a flow diagram of one embodiment of a method to transfer unifiable instructions from the tines to the handle of the graph structure representing the ~~executable~~source code.

[0015] A system and method to reduce the size of ~~executable~~source code in a processing system are described. Multiple subgraph structures are identified within a graph structure constructed for multiple ~~executable~~ source code instructions in a program. Unifiable variables that are not simultaneously used in the ~~executable~~ source code instructions are identified within each subgraph structure. Finally, one or more unifiable instructions from a tine of a corresponding subgraph structure are transferred to a handle of the corresponding subgraph structure, each unifiable instruction containing one or more unifiable variables.

[0016] **Figure 1** is an exemplary ~~executable~~source code within a program. As illustrated in **Figure 1**, in one embodiment, the exemplary ~~executable~~source code 100 includes multiple lines, each line containing a separate ~~executable~~ source code instruction. The ~~executable~~ source code instructions can be at a low backend level, or at some higher level closer to the front end. As shown in **Figure 1**, the ~~executable~~source code 100 includes an "if(p)" clause and an "else" clause, each containing multiple ~~executable~~ source code instructions.

[0017] **Figure 2A** is a block diagram of one embodiment of a graph structure representing the ~~executable~~source code shown in **Figure 1**. As illustrated in **Figure 2A**, in one embodiment, graph structure 200 includes a common predecessor "if(p)" block 201, a subgraph structure containing two tines 202, 203, for example sequences of Instructions with straight-line control flow, also known as basic blocks, and a common successor "return z"

block 204. The basic block 202 contains instructions located in the "if(p)" clause of the ~~executable~~source code 100 and the basic block 203 contains instructions located in the "else" clause. The two basic blocks 202, 203 share two handles, the common predecessor block 201 and the common successor block 204.

[0018] In one embodiment, if a known "omega motion" procedure is used, wherein ~~executable~~ source code instructions are moved downwardly past join points, matching instructions within the basic blocks 202, 203 are unified and subsequently transferred to common successor block 204, while being removed from their respective locations in the basic blocks 202 and 203. Alternatively, if a known "alpha motion" procedure is used, wherein ~~executable~~ source code instructions are moved upwardly past fork points, matching instructions within the basic blocks 202, 203 are unified and subsequently transferred to common predecessor block 201, while being removed from their respective locations within the blocks 202 and 203.

[0020] **Figure 2B** is a block diagram of one embodiment of the graph structure showing a reduced ~~executable~~source code obtained through a method to reduce the size of the ~~executable~~source code in a processing system. As illustrated in **Figure 2B**, the instruction "z = x*y", located within basic block 202, and the instruction "z = a*b", located within basic block 203, are unified by unifying the variables x and b, and unifying the variables y and a. A unified instruction "z = x*y" is transferred to the common successor block 204. The process of transferring the unified instruction to a common handle, for example common successor block 204, will be described in further detail below.

[0021] **Figure 3** is a flow diagram of one embodiment of the method to reduce the size of the ~~executable~~source code. As illustrated in **Figure 3**, at processing block 310, fork subgraph structures are identified within a graph structure, which represents the ~~executable~~source code. In one embodiment, the ~~executable~~source code 100 is scanned for candidate fork subgraph structures. Each fork subgraph structure contains multiple tines or basic blocks, which share a common successor handle or block (in the case of the omega motion procedure) or which share a common predecessor handle or block (in the case of the alpha motion procedure).

[0022] At processing block 320, an interference graph structure is constructed for local variables mentioned on the tines of each subgraph structures. In one embodiment, the interference graph structure indicates which variables of the local variables are simultaneously used in the ~~executable~~ source code instructions within the tines and cannot be unified, i.e., the variables that have overlapping live ranges.

[0023] The interference graph structure is represented as a symmetric matrix INTERFERE (i,j), where i and j correspond to local variables. INTERFERE(i,j,) is true if variables i and j are both live at some point in the program. In one embodiment, determining whether the variables have overlapping live ranges can be achieved using one of many known methods of determination. One of such methods is data-flow analysis. In the exemplary ~~executable~~source code of **Figure 1**, data-flow analysis can determine that a and y are local to different blocks 201 through 204. Another example of such methods is syntactic analysis. Syntactic analysis can determine that b and x are local to their respective blocks, because they are locally scoped.

[0024] **Figure 4A** is a block diagram of one embodiment of an interference graph structure constructed in connection with the ~~executable~~source code. As illustrated in **Figure 4A**, in one embodiment of the interference graph structure 400, node 401 represents local variable a, node 402 represents local variable b, node 403 represents local variable x, node 404 represents local variable y, and node 405 represents local variable z. Variables a and b interfere, respective nodes 401 and 402 being connected to each other. Similarly, variables x and y interfere, respective nodes 403 and 404 being connected to each other.

[0027] Referring back to **Figure 3**, at processing block 330, a data dependence graph structure is constructed for the ~~executable~~source code. In one embodiment, for an alpha motion procedure, the data dependence graph structure has a directed arc u→v, if the instruction u must precede the instruction v, typically because the instruction v uses a value computed by the instruction u, i.e. instruction v depends upon instruction u. Alternatively, for an omega motion procedure, the data dependence graph structure has a directed arc u→v, if the instruction u must succeed the instruction v. An arc u→v is said to be "properly oriented" within a tine t if instructions u and v belong to tine t, and if, when performing

alpha motion, instruction u precedes v, or when performing omega motion, instruction u succeeds instruction v. (An improperly oriented arc can arise from loop-carried dependencies.)

[0028] **Figure 5** is a block diagram of a data dependence graph structure constructed in connection with the ~~executable~~source code. As illustrated in Figure 5, the data dependence graph structure 500 shows the directed edges between any two instructions within the exemplary ~~executable~~source code 100. For example, considering an alpha motion procedure, the foo(&x) instruction must precede the z=x*y instruction, the y=3 instruction must precede the z=x*y instruction, the a=3 instruction must precede the z=a*b instruction, and the foo(&b) instruction must precede the z=a*b instruction.

[0030] **Figure 6** is a flow diagram of one embodiment of a method to transfer unifiable instructions from the tines to the handle of the graph structure representing the ~~executable~~source code. As illustrated in **Figure 6**, at processing block 610, for each tine of a subgraph structure, for each instruction in the tine, dependence arcs are counted and a flag is initialized for each instruction that may be transferred. One embodiment of processing block 610 is implemented in pseudo-code as follows. If all elements of COUNTER are assumed to be previously initialized to zero:

Please amend the Abstact as follows:

A system and method to reduce the size of ~~executable~~ source code in a processing system are described. Multiple subgraph structures are identified within a graph structure constructed for multiple ~~executable~~ source code instructions in a program. Unifiable variables that are not simultaneously used in the ~~executable~~ source code instructions are identified within each subgraph structure. Finally, one or more unifiable instructions from a tine of a corresponding subgraph structure are transferred to a handle of the corresponding subgraph structure, each unifiable instruction containing one or more unifiable variables.